Corbit

Table of Contents

Description of Program

3 - Files
4 - Functions
7 - Constants
9 - Variables
11 - Structures

12 **Instructions**

17 **Next Steps**

18 **Reflection**

Files

Filename	Description
Corbit.cbp	The code::blocks project file, contains index of all files used in compiling corbit. All other files that start with Corbit (capital C) are generated by codeblocks while using this
corbit.cpp	The int main() source file, also contains functions for running program, and the like
classDeclarations.cpp	Declares all classes
parseDeclarations.cpp	Declares overloadable parse functions
entities.txt	Datafile read by program, contains all data for all entites. More detail inside.
LICENSE.txt	Contains license information for corbit
ChangesLog.txt	Generated by autoversioning plugin, contains just about all changelogs
readme.md	The readme file, for when I sync with github
bin	Where binary files are generated to
obj	where object files are generated to
bin/alld42.dll, bin/alleg42.dll, allp42.dll	dll files that must be in the same path as the executable, for allegro

Functions

namespace::name	Description
CYCLE	Timer function for timing calculations
FPS	Timer function for finding how long since last frame
INPUT	Timer function for timing input
changeTimeStep	Changes how fast CYCLE is called, step multiplies previous rate to obtain new rate. Currently not implemented, because you can increase the cycleRate to faster than your computer can handle, crashing Corbit
input	Parses input
drawBuffer	Draws the contents of the buffer to the screen
drawDisplay	Draws everything to the buffer
debug	Draws debug information
initialize	Calls initializeAllegro and initializeFromFile, additional initialization
initializeAllegro	Initializes allegro stuff, sets repeat rate for timer functions, etc.
calculate	Does all gravity calculations, other calculations
cleanup	Deletes everything to avoid memory leaks
physical_t::a, b	Gets on-screen position of physical, I put this in the main cpp file to avoid having to do stuff with extern bitmaps and cameras
physical_t::draw, solarBody_t::draw,	Draw functions for different objects. Same as a and b, it would've been more of a hassle, and more bloated, to have these in entity.cpp

ship_t::draw, habitat_t::draw

viewpoint_t::viewpoint_t	Constructor for viewpoint_t
viewpoint_t::zoom	Changes zoomLevel, passing a negative value decreases it
viewpoint_t::actualZoom	An exponent of zoomLevel, used for other zoom calculations (i.e. drawing)
viewpoint_t::panX	Increases Vx in passed direction
viewpoint_t::panY	Increases Vy in passed direction
viewpoint_t::shift	Moves camera at Vx, Vy speed
viewpoint_t::updateSpeed	When locked onto an entity, updates with that entity's speed
display_t::drawGrid	Draws a grid of dots. No gravity distortion yet, that will come in time
display_t::drawHUD	Draws HUD
physical_t::move	Moves physical at Vx, Vy speed
physical_t::turn	Turns physical
physical_t::totalMass	Gets total mass of physical
ship_t::totalMass	Gets total mass of ship (mass + fuel)
physical_t::acc	Accelerates physical
physical_t::accX	Accelerates physical along x axis
physical_t::accY	Accelerates physical along y axis
Physical_t::orbitV	Gets needed orbital velocity at current height above target
physical_t::Vcen	Gets centripetal velocity in relation to target
physical_t::Vtan	Gets tangential velocity in relation to target

Physical_t::Vtarg	Gets velocity relative to target
physical_t::thetaV	Gets theta of physical's velocity vector from x axis
physical_t::thetaToObject	Gets theta between physical and target
physical_t::distance	Gets distance from physical to target
physical_t::gravity	Gets gravitational acceleration to target
physical_t::gravitate	Accelerates towards target, with gravitational acceleration
physical_t::detectCollision	Detects collision with physical and target, and bounces them. I haven't completely tested this yet.
physical_t::move	Moves physical
ship_t::move	Calls fireEngine, then moves ship
ship_t::fireEngine	Fires engines at enginePercent * enginePower, uses enginePercent * burnRate kgs of fuel
ship_t::eccentricity	Calculates eccentricity, only accounting for target object, haven't fully implemented
initializeFromFile	Reads entites.txt, parses data, only for solar objects. The last section that is commented out will read background stars
parse, parseColor	Parses data from stream into data variable, returns false if failed. Overloadable for different data types.

Constants

namespace::name	Description
screenWidth, screenHeight	Size of screen on startup, minus taskbar
FPS_COUNT_BPS, INPUT_BPS	Times per second fps, input is checked
AU/au	One Astronomical Unit, different cases to prevent name conflicts
PI/Pi	Pi. Again, to prevent name conflicts
G	Newton can explain this one (gravitational constant)
entity_enum	For ease of selecting specific entities. Must match order entities are read in from file (for now, until I figure out efficient STL searching)
viewpoint_t::zoomMagnitude	The exponent zoomLevel is raised by (see viewpoint_t::actualZoom())
viewpoint_t::zoomStep	The step zoomLevel is increased by when zooming in/out
viewpoint_t::panSpeed	The rate at which the camera's speed is changed
viewpoint_t::maxZoom	How far in you can zoom
viewpoint_t::minZoom	How far out you can zoom
display_t::gridSpace, lineSpace	Pixels in between each dot on the grid, and each line on the HUD
display_t::targVectorLength, vVectorLength	Magnitude of target orientation vector, velocity vector
display_t::craftX, craftY	Position habitat is drawn on screen (craftY is constructed as x * lineSpace)
physical_t::name	Name of physical

physical_t::radius	Radius of physical, in m
physical_t::mass	Mass of physical, in metric tonnes
physical_t::fillColor	Main color of physical
physical_t::atmosphereHeight	Height of physical's atmosphere, in m, if it is used as a solarBody_t
physical_t::atmosphereDrag	Atmospheric drag of solarBody_t. Don't actually know how to implement this yet
solarBody_t::atmosphereColor	Color of atmosphere of solarBody_t
ship_t::engineColor	Engine color of ship
ship_t::engineRadius	Radius of engines
ship_t::enginePower, burnRate	Power of engines in newtons, and how fast they use fuel
debug::spacing	How far down the screen to start printing debug information

Variables

namespace::name	Description
bool printDebug	Whether to print debug information
unsigned long long cycleRate	How many calculations to perform per second (WARNING: don't set too high!)
volatile unsigned int cycle, cps, cycleCounter, fps, fpsCounter, inputTimer	Timing and counting calculations performed, counting fps, timing input
vector <physical_t*> entity	Vector all entities are stored in
vector <physical_t*>::iterator it, itX, itY	Iterators for calculations, publicly declared so they aren't redeclared literally thousands of times a second
viewpoint_t::x, y, Vx, Vy	Position/velocity of camera along axes
viewpoint_t::zoomLevel	Not actual zoom of camera, but change this to change zoom
viewpoint_t::struct physical_t *target, *reference	Pointers to target and reference of camera. Reference doesn't actually do anything (yet)
viewpoint_t::bool track	Locks camera on target
display_t::struct physical_t *craft, *target, *reference	Pointers to the craft displayed on HUD, target, and reference (again, doesn't do anything YET)
physical_t::long double x, y	Position of physical on axes
physical_t::float turnRadians	Radians physical is rotated from x axis

physical_t::long double acceleration	Total acceleration of physical
physical_t::long double Vx, Vy	Velocity of physical along axes
physical_t::long double turnRate	Rate at which physical is turning, in radians
physical_t::float turnRateStep, maxTurnRate	Rate at which turnRate is changed, highest turnRate, in radians
physical_t::AI_t::navmode	Enum identifying current navmode (haven't actually implemented different navmodes yet)
physical_t::AI_t::descriptor	String describing current navmode
physical_t::engine	Percent engines are burning at, if used as a ship_t. Just like original orbit, they can go negative or over 100%
physical_t::fuel	Kgs of fuel left

Structures

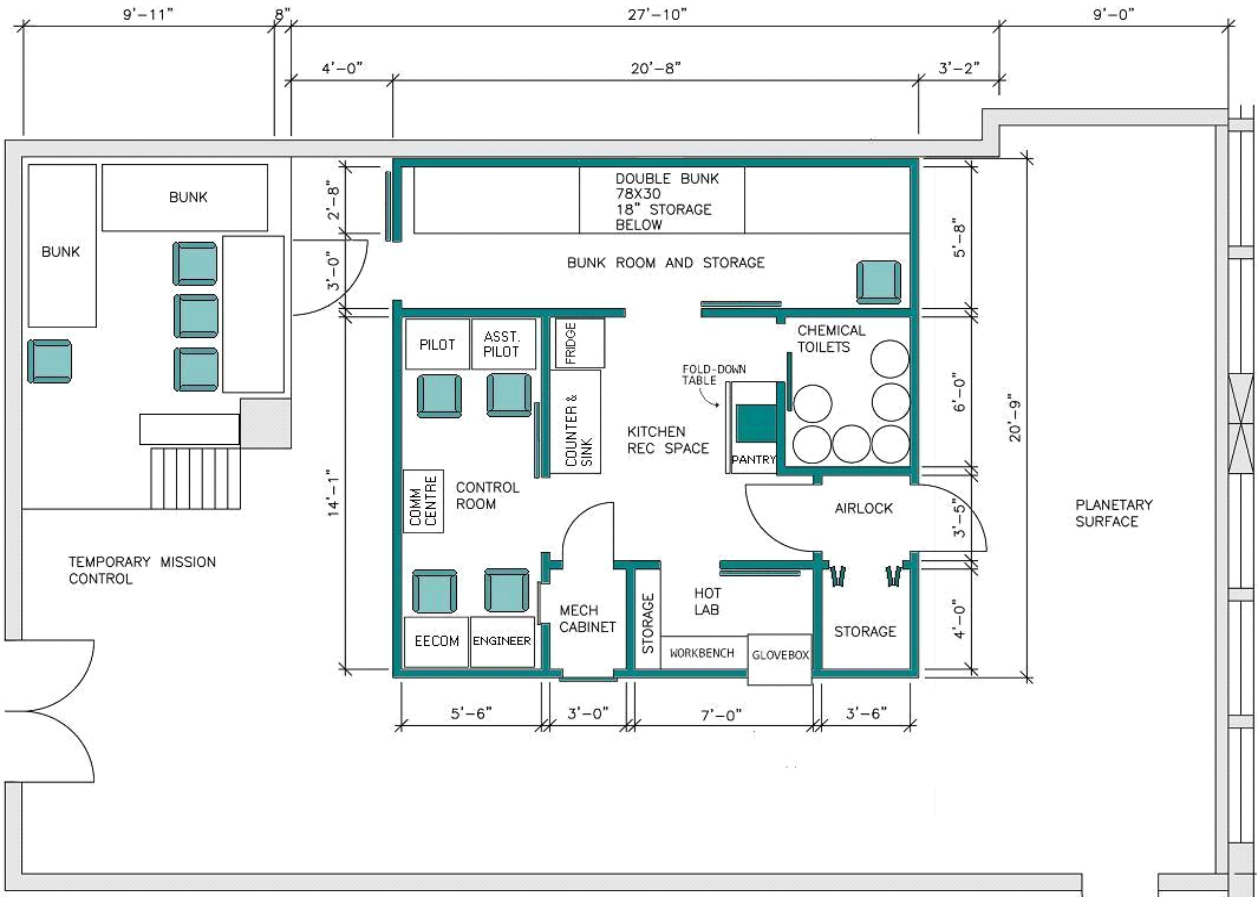
Name	Example	Description
viewpoint_t	camera	Stores data pertaining to camera position, panning, zoom level, etc.
display_t	HUD	Prints out grid, HUD and does mathematical calculations for a bunch of stuff related to the HUD
physical_t		Default, base class for any entity
solarBody_t : physical_t	entity[EARTH]	Derived class for a solar body (planets, moons, asteroids, etc. go in here)
ship_t : physical_t		Default, derived class for a ship. Specific ships/space stations only need to have a redeclaration of virtual void draw() as void draw(), then a custom method for drawing it
habitat_t : ship_t	entity[HAB]	Derived class for the habitat. Uses custom void draw()

Instructions

There's a common analogy: When something is simple and mundane, we compare it to driving a car. When something is more complex, it is compared to flying a state-of-the-art jet fighter.

You are flying a pseudo-science interplanetary multiple atmospheric exit and re-entry vehicle.

It is called the Hawking III, or the Habitat (hab). On it is enough materials for six brave astronauts to live together for a week, control the hab, monitor its atmosphere, life support, power grid, engineering, life support, refrigeration of rations and fuel, energy production (deuterium fusion reactors are fun), and still have room to perform experiments on anything they deem in the interest of science.



Hawking III blueprints

When you first use this simulator, you will not know what you are doing. Second time, maybe a bit more, but you still not know how to do it. People train on these simulators and get good at it after

months, even with just two dimension of space. So for simplicity's sake, there are only two dimensions of space. The simulator has full control over all systems, and you cannot simulate anything. It cannot communicate with engineering, or EECOM, or the simulator's software, because I don't expect one person to do the job that six people usually do over the course of a mission. That would be silly. So the program is much simpler than it would be.

To achieve orbit, you must first calculate the required speed (Kepler's third law, find a calculator on the internet), then make sure your Vtan around the target it equal to that speed, and Vcen is equal to zero. This will put you into an orbit around the target, assuming it is a circular orbit (which it should be).

Note: don't try to go too close to planets, on slow/medium computers it will lag a bit. Same goes for zooming out on slow/medium computers (but only the first time you zoom out) as the computer must render all objects for the first time.

HUD

Display name	Description
Orbit V	Required orbital velocity (not implemented yet)
Hab/Targ V diff	Difference in velocity between habitat and target
Centripetal V	Centripetal velocity in relation to target
Tangential V	Tangential velocity in relation to target
Fuel	How many kgs of fuel left
Engines	Percent engines are at (can be more than 100 or less than 0)
Acc	Acceleration of habitat
Turning	Rate of turning in degrees/s
Altitude	Distance from surface of target
Pitch	Pitch angle of habitat in relation to target (not implemented yet)

Stopping acc	Acc required to stop at target (I don't actually know whether this works, I just took the algorithm from orbit)
Periapsis	Lowest point of orbit (not implemented yet)
Apoapsis	Highest point of orbit (not implemented yet)
(habitat)	Draws velocity vector and position relative to target on HUD hab
Target	The name of the target
Reference	The name of the reference
Autopilot	The description of the current navmode
FPS	Current FPS
Timestep	How many times normal speed program is calculating

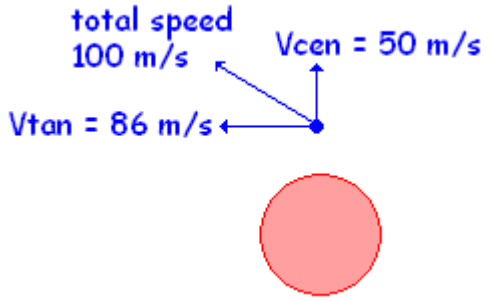
Keys

Key	Result
W, S	Throttle up/down by 1%. Shift modulates this to 0.1%
A, D	Use RCS thrusters to turn CCW/CW. Only works in manual navmode
TAB	Toggles camera tracking of the current camera's center
ARROW KEYS	Pans camera (when in free camera mode)
+, -	Zooms camera in/out
~	Toggles HUD
Z	Toggles printing of debug information

G	Toggles grid display
BACKSPACE	Cuts engines
CTRL + BACKSPACE	Automatically uses RCS thrusters to stop turning
ENTER	Engines to 100% cap'n!
ALT + +/-	Increase time acceleration **WARNING:** Do not set too high or the program will hang. It may be able to go up to 307445735000000000, but don't do that
0 – 9, H	Set target as nth planet from the sun (sun is 0, hab is H)
SHIFT/ALT + 0 – 9	Set target/reference as nth planet from sun (sun is 0)

Physics

Motion in space:



In space, all motion is relative. While the program uses a Cartesian system wherein all absolute velocities and positions in the program are relative to the Sun, we have to use temporary reference points for anything that we do while piloting. Corbit calculates relative velocities like this. In the example on the right, the total speed relative to the object is 100 m/s. The only times when Vcen and Vtan should be almost zero is on landing or in a circular, stable orbit. For other piloting maneuvers, the two should be considered differently. Vcen (centripetal velocity) is the velocity you are moving away from the object, and can be described as negative (going towards object) or positive (away). Vtan (tangential motion) is the side-to-side velocity. This can be described as CCW or CW. The usual Vtan is a CCW one, like just about all solar bodies.

NOTE: ALL movement must be accomplished using engines. Speeding up and slowing down are obvious ones, but changing direction as well.

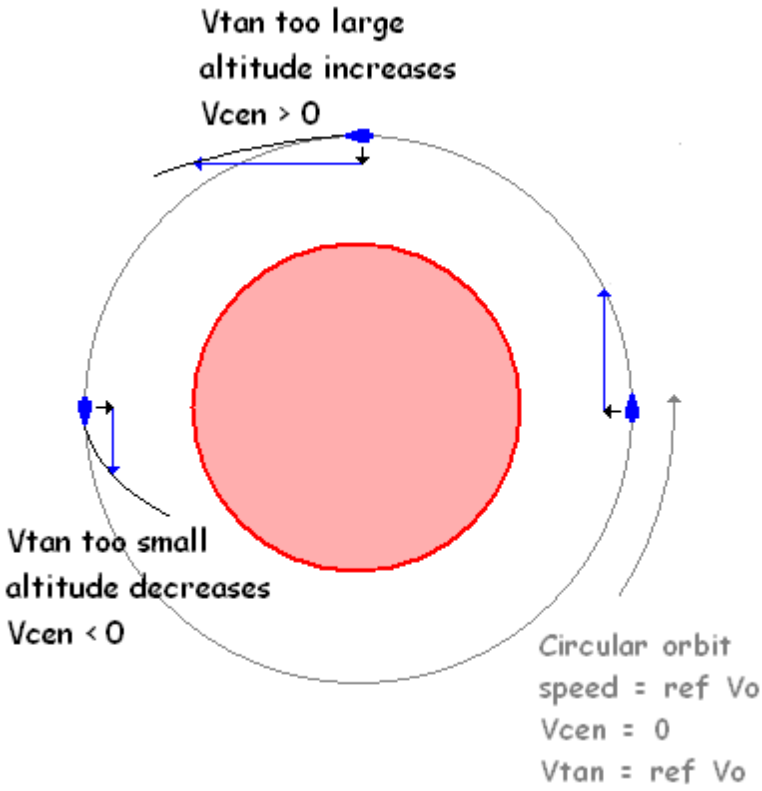
Motion in orbit:

To obtain a stable orbit, or a perfectly circular orbit, the following conditions must be met:

- Hab/Target V = Vtan
- Total speed = Hab/Target V
- Vcen = 0

If any of these conditions are not met, the orbit will be elliptical. Obviously, all orbits are at least very slightly elliptical, but the farther off you are, the more elliptical the orbit.

And those are the basics. Visit spacesim.org for more documentation, physics explanation, and the like (documentation is for original orbit, however).



Next Steps

Program Additions

- Optimize drawing so that there are no lag spikes, maybe also have gradients for an atmosphere
- Optimize gravity calculations so that, for example, Pluto doesn't affect the Sun, but not vice versa
- Balance Corbit values to exactly match those of orbit (e.g. fuel burn rates, engine power)
- Implement Corbit/orbit suite program communication (i.e. communication with EECOM, ENG, maybe TRANSORB, ORBIT5T save files)
- Add AYSE, ISS, other solar bodies (i.e. all entities in orbit, since I have a vector of entities, this will just *happen* when I can read ORBIT5T save files)
- Optimize time dilation, by adding approximations and extrapolations
- Cool graphics, possibly raster graphics (e.g. fuel tanks, HUD design, Habitat, small objects, MAYBE for large objects too?)
- More realistic physics, maybe particle physics???
- Animations possibly?

Code Additions

- Look into smart pointers, boost library (which I've already done, but for simplicity of reading code's sake, I only use dumb pointers)
- Add more classes for different objects (SRB debris, fuel tank debris, etc.)
- Look into different graphics libraries for drawing operations (openGL sounds cool)
- More file splitting, maybe calling functions?

Reflection

So here we are.

After 73 days of development, and several sleepless nights near the end, I'm finally at the end of the end. And it's a very gratifying experience. Reasons why it was gratifying:

- I learnt exactly how orbit (the program and the gravity one) works, and how to maneuver properly
- I self-taught myself vector manipulation (see physical_t::detectCollision), which will definitely come in handy for physics and more Corbit coding
- I figured out a good deal of orbital mechanics, how to calculate relative velocities, and do all sorts of velocity and vector manipulation like that
- STL vectors are now a thing I know how to do now, which are better for memory management, and also iteration and stuff like that, and is easier to read different amounts of entities into the program
- While I didn't actually implement it, because I based my code on structs, I know how to use calling functions for all my class-defined variables, and declare my variables as privates, which I will definitely start out doing for my next project, because that is good programming structure.
- Splitting code into files makes the main source code much more concise, and makes it easier to modify source code, as you don't have to scroll through thousands of lines of code
- Using a good IDE (e.g. CodeBlocks) is definitely useful. Auto-format has saved my sanity on multiple occasions, and the colorful syntax highlighting has made the code much more readable. Also, CodeBlocks is much more customizable than Dev C++, allowing me easier control over how I format my code, my keyboard shortcuts, and autoversioning
- Spending less time on trying to add in not-necessarily essential features, such as boost::shared_ptr, which I spent a few days on, and boost::zip_iterator, which I found out I had been using incorrectly, and I spent a week trying to implement. Also, it gets kind of annoying to

work on the same problem for so long, so spending less time on a problem would be a more efficient use of time.

- Streams, and how useful they are. Cout is definitely useful, but ifstream, isstream, and all those streams are just so much easier to work with, and are very flexible
- VCS and how useful they can be. I don't think I would've been able to be this efficient without GitHub and its great bash command line